VMlib – an object-oriented library for visualizing observation sequences in spacecraft missions

Ansel Teng and Meemong Lee*
Jet Propulsion Lab
4800 Oak Grove Dr. MS 168-522
Pasadena, CA 91109

Keywords: *visual simulation, spacecraft missions, observation sequence, interactive 3-D graphics, object-oriented programming.*

Abstract

VMlib is a C++ library for visualizing spacecraft mission simulations with an emphasis on the execution of observation sequences. It provides multiple views on the scenario using the state-of-the-art interactive 3-D computer graphics to help the design and validation of the observation sequence. It is designed to be a flexible toolkit to support multiple missions in a variety of application environments. Mission independence is achieved by parameterizing mission description into input files. The object-oriented design allows the mission class to be subclassed for various application environments. The library has been used to develop both stand-alone programs and a visualization server in distributed simulations. These applications are used in support of several on-going missions at the Jet Propulsion Lab.

1 Introduction

Visualization is a powerful tool to convey complex concepts and to understand complex phenomena. With the increasing graphics capability in the desktop workstations, visualization has been used in several commercial software packages [1, 2, 6] to present simulation results for spacecraft mission design. However, the existing software packages focus on either trajectory design or ground coverage analysis, and provide little help for the design of the observation sequences in a spacecraft mission. An observation sequence consists of a sequence of commands that orchestrates the subsystems of the spacecraft so that the instrument subsystem can optimally observe the target to obtain the best science data possible. Traditionally, observation sequences are designed and validated based on mathematical calculations. Without a good comprehension of the 3D geometry involved in the sequence, it is very difficult to design a sequence that maximizes the scientific return while satisfies all the constraints, let alone evaluating the validity of the sequence in the face of uncertainties.

As part of our effort to create a comprehensive simulation system for the design and validation of generic science observation sequences [4], we developed the Virtual Mission Visualization Library (VMlib) to provide the visualization capability of the system. Since our system is designed to support multiple missions, the library must also be mission-independent. Furthermore, each simulation scenario may require a different combinations of program modules, so the library must be self-sufficient to run in a stand alone program yet adaptable to allow connections to other simulation modules in a distributed environment.

The library is written in C++ following the object-oriented paradigm. It consists of a set of classes that provide high-level visualization capabilities for spacecraft mission scenarios. The graphical functionalities are built on top of the Open Inventor 3-D graphics library [5]. Open Inventor contains a comprehensive set of classes and methods that can be used to create interactive 3D graphics applications. It is a high level graphics library in the sense that the user simply builds a scene graph to describe the scene and the viewing camera while the library handles the details of the rendering process. Using the Open Inventor library enables us to focus our development efforts on constructing scene graphs from mission descriptions, interfacing with other mission simulation components, and exploring various visualization methods.

We will present the VMlib class hierarchy in the next section and describe the mission parameters in Sections 3. Application examples are discussed in Section 4. Data created in our visualization for the New Millennium DS-1 mission [3] will be used throughout the paper for illustrations.

*This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

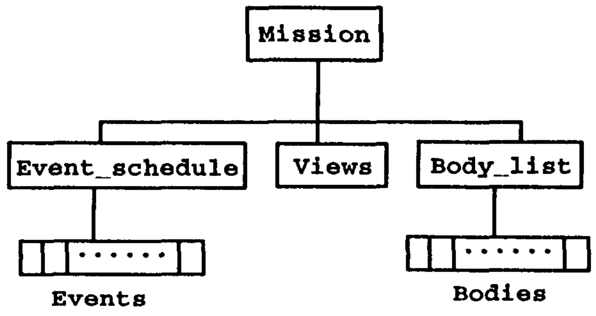

Figure 1: The major classes in the VMlib and their relation.

2 Class Hierarchy

VMlib consists of several object classes as shown in Figure 1. The Mission class sits at the top of the hierarchy. An application instantiates a Mission object to create the visualization. A Mission object initializes the component objects, controls the progress of the simulation, and coordinates the interactions among the components. Under the Mission class, the Body_list maintains the information of the bodies involved in the visualization, such as the spacecraft, the Sun, the Earth, and the target bodies. The Event_schedule is a list of events for visualization control such as starting/ending time, time resolution control, and viewpoint control. Several View class objects visualize simulation results using a variety of viewpoints and methods.

The operation model of the Mission class follows the event-driven paradigm provided by the Open Inventor library. After the initialization, the start() method registers event handlers and enters the main loop. The progress of the simulation is made by the update() method called from a timer event handler. The update() method performs the following tasks:

1. Execute commands in the Event_schedule up to the current simulated time;
2. Update the geometry of each bodies in the Body_list;
3. Re-render the views;
4. Determine the time of the next simulation step.

The timer event handler can be scheduled or unscheduled by a mouse event handler. This allows the user to start/pause/resume the visualization at the click of a mouse.

The following subsections provide more detailed descriptions of the components:

2.1 Body List

The Body_list class maintains an array of Body class objects. Each Body object contains attributes of its geometry and appearance such as position, attitude, size, shape, solid model, color, texture, etc. An Orbit class object can also be attached to a Body object to show the trajectory of the object.

Most of the attributes are set during the initialization and remains unchanged throughout the simulation. The position, attitude, and orbit are updated for each simulation step. The *state*, i.e. position and attitude, of the celestial bodies are calculated from trajectory data files. The data files are in the SPICE format invented at JPL for archiving spacecraft states and mission events. The Object-Oriented SPICE (OOSPICE) library [9] is used to propagate a body state for specified time and reference coordinate system. The spacecraft state can be updated either by using a trajectory data file, or by assigning the result from another simulation components.

Choosing coordinate systems is critical in a cosmic-scale visualization [7]. 64-bit precision is required to maintain the accuracy in coordinate calculation, but the Open Inventor library and the underlying graphics hardware supports only 32-bit precision for floating-point numbers. To accommodate these two conflicting requirements, each body maintains two coordinates: a *world coordinate* with respect to a scenario-specific reference body using double-precision numbers for position calculation, and a *graphical coordinate* with respect to the spacecraft for constructing the scene graphs. The world coordinate system provides intuitive interpretations of the spacecraft movement, while the the graphical coordinate system maximizes the depth resolution needed near the spacecraft.

2.2 Views

The VMlib provides multiple views to look at several aspects of the simulation. Currently, four types of views are supported:

1. Trajectory view: visualizes the geometry among the bodies with their trajectories, as shown in Figure 2. It provides an overview of the simulated scenario.
2. Spacecraft view: a close-up on the spacecraft to show its attitude change and articulation. In the example shown in Figure 3, the cylindrical beam indicates the sun light direction and the cone along the instrument field-of-view marks where sun light should be avoided.

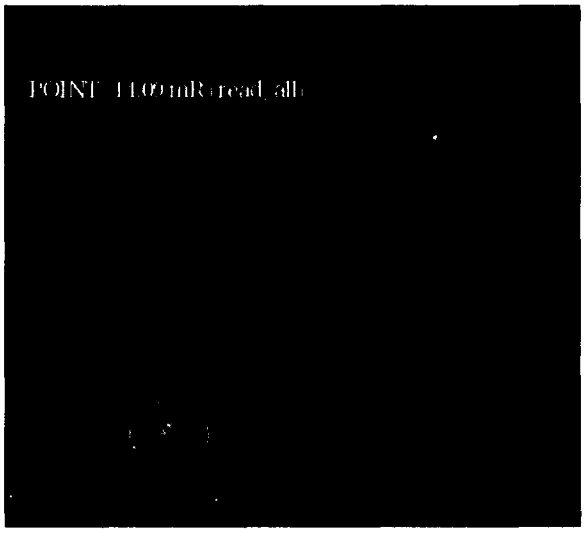

Figure 2: The trajectory view showing the DS-1 spacecraft, the Earth, and the Moon.

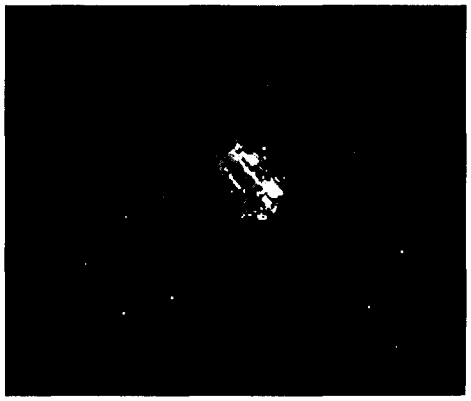

Figure 3: The spacecraft view showing the DS-1 spacecraft.

3. Instrument view: provides the view through the instrument system to visualize the geometry between the target body and the fields-of-view of the sensors. See Figure 4 for an example. The field-of-view indicator of each sensor can be set to blink to indicate a snap action. Although this view is hardly an instrument simulation, it provides critical information regarding the quality of the image product such as target size, target position, and sun light phase angle.

4. Pointing view: an abstraction of the instrument view with a further reduced field-of-view. It visualizes the minute movement of the target with respect to the instrument system at sub-pixel resolution, as shown in Figure 5. The target position is marked by an arrow pointing along its direction of motion. Its trajectory in the view is indicated by line seqments that fade over time, and grid lines are drawn to provide the scale. This view is useful for checking the pointing accuracy and the stability of the spacecraft, and predicting their impact (such as blurring) on the data product quality.

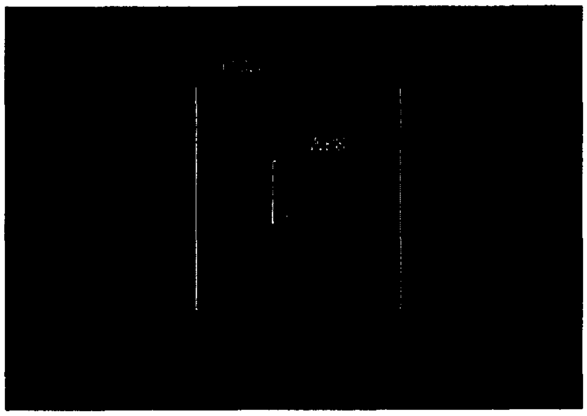

Figure 4: The instrument view showing the Earth and the fields of view of the four sensors.

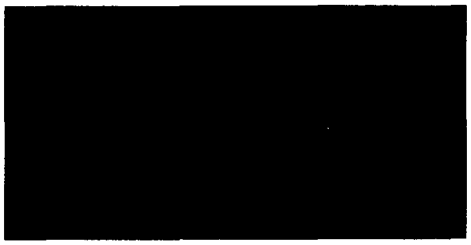

Figure 5: The pointing view showing the target movement with regard to the instrument boresight during an IR scan operation.

The View classes are designed using inheritance: a base VM_View class provides the common functionality among all views, while a derived class is created for each view type to handle the specifics. Although all views are used to look at the geometric relation maintained in the Body_list, each view constructs its own scene graph so that the bodies appear in each view with appropriate scales and features. For a realistic visual effect, star field can be added to the background using data from a real star catalog.

2.3 Event Schedule

The Event_schedule class maintains a list of Event class objects. An Event object is a visualization control command that should be performed at a specific time.

The command may be a progress control such as stopping the simulation or changing the time resolution, or a graphics control such as displaying a label or changing the viewing parameters.

Since the time at which a particular event should be executed is usually related to another event such as launch, the start/end of an observation sequence, or the closest encounter with a target, the time attribute of an Event is implemented using a base-offset mechanism. The Event_schedule maintains a table of base times, and each Event object contains a pointer to an entry of the base time table and an offset with respect to that base time. With this mechanism, the same event description file can be used for testing an observation sequence at different times.

3 Mission Parameters

The VMlib is designed to support multiple missions. Consequently, the mission data must be parameterized and read as input during run time. Since a mission description requires numerous parameters, they are maintained in files. The parameter file syntax is defined in a way such that the parameters can be grouped hierarchically to reflect the object structure and the relation among the objects. A Mission object will then be constructed from such a parameter file. For the reason of modularity, the Body_list and the Event_schedule will be constructed from their own parameter files and only the file names will be kept in the Mission parameter file.

The following subsections show parameter file examples for a calibration sequence of the DS-1 mission. The sequence is scheduled at the 29th day after the launch using either the Earth or the Moon as the target.

3.1 Mission

The following is the Mission class parameter file:

```
NAME = NEWM DS-1
MISSION_CONFIG = ds1.mission
BODY_LIST = newm.body_list
EVENT = newmL29.event
REFERENCE = EARTH
TR_VIEW {
    SIZE = 640 540
    USE_VIEWER = 1
}
SC_VIEW {
}
IS_VIEW {
    FOV_MAP = micas.fovmap.iv
    SIZE = 660 660
}
BACKGROUND = 0.0 0.0 0.
STAR_FIELD_SIZE = 6000
```

The first few line provides the name of the mission and the names of other parameter files for initializing the component objects. The MISSION_CONFIG parameter is the file name of the mission configuration file which contains the basic information of the mission and the observation sequences. The REFERENCE keyword specifies that the earth is used as the origin for the world coordinate system. The View class objects are specified subsequently and the attributes for each view are grouped using a pair of curly braces under the name of the view. The last two lines specifies a black sky with a 6000-star star field as the background of the views.

3.2 Body list

The parameter file for the Body_list object is shown as follows:

```
NEWMILLENNIUM {
    TYPE = SC
    SCALE = 60
    SHAPE = ds1_sc.iv
    ORBIT = red
}
SUNB {
    TYPE = STAR
}
MCAULIFFE {
    TYPE = ASTEROID
    SCALE = 40000
    COLOR = seagreen
    TEXTURE = dion2.rgb
    ORBIT = seagreen
}
EARTH {
    TYPE = PLANET
    SCALE = 120000
    COLOR = skyblue
    TEXTURE = clouded_earth_b_256.rgb
    ORBIT = skyblue
}
```

Four bodies are listed in this file: the Sun, the Earth, a spacecraft named NEWMILLENNIUM, and the target asteroid named MCAULIFFE. The attributes grouped under each body name describe the visual property of the body.

3.3 Event Schedule

The following is the Event_schedule parameter file for this scenario:

```
SECTION {
  NAME = PRE_SEQ
```

```
    T_BASE = JUL-30-1998/12:00:00
    EVENTS {
      JUL-13-1998/00:00:00 start
      JUL-13-1998/00:00:00 label L+29 Sequence
      JUL-13-1998/00:00:00 sc_pointing  MOON
      JUL-29-1998/00:00:00 time_inc 3600
      JUL-30-1998/11:59:50 time_inc 1
      JUL-30-1998/12:00:00 label Sequence Start
    }
  }
  SECTION {
    NAME = POST_SEQ
    T_BASE = JUL-30-1998/13:59:00
    EVENTS {
      JUL-30-1998/14:00:00 time_inc 86400
      AUG-30-1998/00:00:00 stop
    }
  }
```

The event schedule is divided into two sections: the first one, named "PRE_SEQ", is intended to control the visualization before the observation sequence, and the second one, named "POST_SEQ", controls the visualization afterward. The T_BASE parameter provides the base time for each section. Although each event is described using an absolute time, the section base time will be subtracted from the event time to obtain the offset. The Event_schedule will maintain a table containing the two base times and set the base time pointer in each event to the appropriate entry in the table.

3.4 Parsing Parameter Files

Although each parameter file has its own set of keywords and arrangements, they can all be modeled as a tree structure in which each node contains a pair of keyword and value strings. To facilitate efficient parameter retrieval from the structure, we created another class library, called Partree [8], to parse a parameter file into such a tree structure. Instead of reading the parameter file directly, each object will be constructed from a tree or a branch (subtree) using the search and navigation methods provided by the Partree class. Using this intermediate structure has a few advantages:

1. Reduced development cost: instead of writing a complete parser for each class, the effort spent in parsing sophisticated lexical and syntactical rules is re-used;

2. Better control in interpreting input: the program actively searches for parameter keyword and takes action upon the search result. This usually improves the code structure;

3. Enables sharing parameter files between a base class and its derived class: if both the base class and the derived class are constructed from a Partree object, the derived class constructor will be able to initialize its base class using the same Partree object.

4 Applications

The Mission class discussed above is capable of performing a visual simulation on its own. The spacecraft position will be interpolated from the OOSPICE data file and the attitude will be set to point the spacecraft instrument at the target automatically. This is useful for getting an overview of the trajectory as well as for understanding the target size and phase angle in the data product.

The VMlib is designed to support not only multiple missions, but also a variety of application environments. The adaptation to a specific application environment is done by using class derivation. The following subsections describe the applications developed based on the VMlib.

4.1 Observation Sequence Design

The objective of this application is to provide a stand alone program for the sequence designer to visualize the execution of the observation sequence in the context of spacecraft operation. The execution of an observation sequence includes turning the spacecraft to have a specific instrument pointed at the target or slewed across the target. It is necessary to integrate with our payload executive module [4] in order to simulate the spacecraft pointing and navigation mechanism.

To incorporate the payload executive module into the VMlib, we derived a Mission_pl class from the base Mission class. The following functions are implemented to override those of the base class:

1. The constructor: like the base Mission class, a Mission_pl class is initialized from a Partree object. The constructor initializes the base class using the same partree, and retrieves the parameters that are added specifically for the derived object. These additional parameters are:

```
SEQUENCE = sequence29
EVENT = newmL29.event {
    T_BASE_REF {
        PRE_SEQ = <BOS>
        POST_SEQ = <EOS>
    }
}
```

The Mission_pl object constructs a payload executive object using the SEQUENCE parameter which

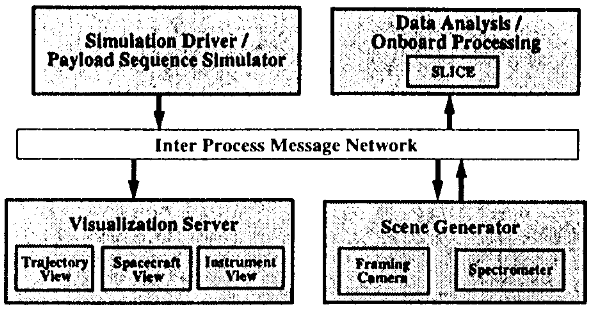

Figure 6: The distributed simulation architecture.

specifies a sequence in the mission configuration file to simulate. The T_BASE_REF attributes specifies how to adjust the event base times according to the sequence. The symbol <BOS> refers to the beginning of the sequence and <EOS> the end of the sequence. These times are available from the payload executive module. The base time table in the Event_schedule will be modified so that the simulation events are scheduled according to the actual sequence time.

2. The update() method: instead of asking the Body_list to update spacecraft states using a trajectory file, update() now calls a method of the payload executive object to obtained the simulated spacecraft state and sets the corresponding data in the Body_list.

With the Mission_pl class, the sequence designer can interactively edit the observation sequence and see the simulation result. The same sequence can also be applied to a different target or at a different time without changing the visualization parameter files. Spacecraft performance parameters can be modified and navigation errors can be injected to test what-if scenarios and to evaluate the impact of the spacecraft performance on the image product quality.

4.2 Instrument Performance Simulation

In a large scale instrument performance simulation that involves not just the spacecraft system but also the camera simulation and data analysis tools, running all components under one program becomes impractical. Instead, a message-driven distributed simulation is used as shown in Figure 6.

In this architecture, the VMlib is adapted as a visualization server that reacts to messages from the simulation driver. The driver maintains the event schedule (by using a Event_schedule object) and the payload sequence to determine the current simulation time, calls the payload executive module for spacecraft states, and sends out messages to the other modules.

To adapt VMlib for this environment, we derived the Mission_sv class from the base Mission class. The constructor is identical to the Mission class. When a Mission_sv object starts its main loop, instead of inserting a timer event handler for update, it registers a remote procedure call (RPC) request handler to execute the incoming events. A small relay program sits between the visualization server and the inter-process message network to translate the messages into Event objects and make RCP calls to the visualization server.

5 Conclusion

In this paper, we presented VMlib, an object-oriented library for visualizing the execution of observation sequences in spacecraft missions. The library is designed to be a flexible toolkit to support multiple missions in a variety of application environments. The versatility is achieved through object-oriented design, mission parametrization, and the use of an intermediate structure for parsing parameter files.

The library provides multiple views for visualizing the observation sequence to help understanding the impact of spacecraft performance on the instrument observations. It has been used in several of the advanced mission designs at the Jet Propulsion Lab, and has facilitated efficient identification of problem areas during the early phase of a mission design process. It is a solid building block toward the faster-cheaper-better mission design methodology.

References

[1] Analytical Graphics. 1997. Satellite Tool Kit. *http://www.stk.com.*

[2] Autometric Incorporated. 1997. OMNI. *http://www.autometric.com.*

[3] Jet Propulsion Lab. 1997. Deep Space One. *http://epic.jpl.nasa.gov/ds1.*

[4] M. Lee, R.L. Swartz, A. Teng, and R.J. Weidner. Encounter geometry and science data gathering simulation. In *Proc. AIAA Guidance and Control Conference*, 1997.

[5] Open Inventor Architecture Group. 1994. *Open Inventor^{TM} C++ Reference Manual.* Addison-Wesley.

[6] D.Y. Stodden and G.D. Galasso. 1996. *Satellite Orbit Analysis Program (SOAP) User's Manual.* The Aerospace Corporation.

[7] M.R. Stytz, J. Vanderburgh, and S.B. Banks. 1997. The Solar System Modeler. *IEEE Computer Graphics and Applications*, 17(5), pages 47-57.

[8] A. Teng. 1997. PARTREE - a parameter file parser for application and object initialization. *http://houyi.jpl.nasa.gov/ teng/partree.*

[9] R.J. Weidner. 1997. Object-oriented SPICE Library. *http://cicero.jpl.nasa.gov/richard.*